社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2006－SLDM－124 (6)
2006／3／16

拡張性及びオーバーヘッドを考慮したRMT Processor用リアルタイムスケジューラの設計と実装

加藤 真平† 小林 秀典† 山崎 信行†

† 慶應義塾大学大学院理工学研究科開放環境科学専攻
〒223-8522 横浜市港北区日吉3-14-1
E-mail: †{shinpei,kobayashi,yamasaki}@ny.ics.keio.ac.jp

あらまし 本論文では，RMT Processorを対象としたリアルタイムスケジューラの設計及び実装について述べる．スケジューリングアルゴリズムには，U-Linkスケジューリング方式のアルゴリズムであるUL-RM及びUL-EDFを利用する．RMT Processorを含むマルチスレッドプロセッサでは，スケジューリングアルゴリズムの性能がタスクセットに大きく依存する．そこで，複数のスケジューリングアルゴリズムにも対応できるように拡張性を考慮した設計を行う．また，マルチスレッドプロセッサでは，ユニプロセッサに比べてタスク管理のオーバーヘッドが大きくなる傾向がある．そこで，計算量の少ないタスク管理機構の実装を行う．さらに，U-Linkスケジューリング方式の特徴を有効利用し，グローバルスケジューリング方式やパーティショニング方式に比べて，スケジューラ呼び出しによるオーバーヘッドを削減できるように実装を行う．

キーワード リアルタイムスケジューリング，U-Linkスケジューリング，タスクスケジューラ，RMT Processor，SMT

Design and Implementation of Real-Time Scheduler with Expandability and Low Overhead for RMT Processor

Shinpei KATO†, Hidenori KOBAYASHI†, and Nobuyuki YAMASAKI†

† Keio University
3-14-1 Hiyoshi, Kouhoku-ku, Yokohama, Kanagawa 223-8522 Japan
E-mail: †{shinpei,kobayashi,yamasaki}@ny.ics.keio.ac.jp

Abstract This paper describes design and implementation of the real-time scheduler for RMT Processor. We make use of UL-RM and UL-EDF which are the algorithms of U-Link Scheduling scheme. On multithreaded processors, including RMT Processor, the effectiveness of scheduling algorithms depends on a task set. We then design the scheduler so as to have an expandability to a lot of scheduling algorithms. Also the overhead of task management on multithreaded processors tends to be larger than that on uniprocessors. Therefore, we implement the task management mechanism which does not require heavy computation. In addition, we impelement the schedule function whose overhead, caused by the scheduler-calls, is lower than the global scheduling scheme or the partitioning scheme, by making use of the advancement of U-Link Scheduling scheme.

Key words Real-Time Scheduling, Task Scheduler, U-Link Scheduling, RMT Processor, SMT

1. はじめに

近年の組み込みシステムでは，リアルタイム性だけでなくシステム全体のスループットも要求される．しかしながら，ハードウェア資源や消費電力に制限がある組み込みシステムでは，動作周波数を上げることが難しい．そういった中で，動作周波数ではなくプログラムの並列度を高めることでスループットを向上するマルチスレッド実行が注目されている．とりわけ，複数のスレッドを同時に実行できるSimultaneously Multithreading(SMT) [7]は，従来のスーパースカラや細粒度マルチスレッディングに比べて2~3倍のスループットを達成できることが知られている[3]．我々の研究室においても，各スレッドの優先度に従ってSMT実行が可能な並列分散リアルタイム処理用プロセッサであるResponsive Multithreaded Processor(RMT

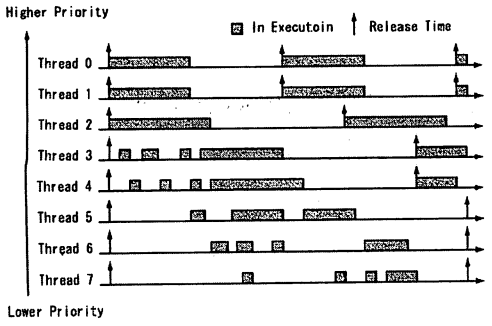

図 1　RMT Processor における実行の様子

Processor) [8] の研究開発を行っている．

マルチスレッド実行が可能なプロセッサが主流になったことにより，近年，マルチプロセッサやマルチスレッドプロセッサを対象としたリアルタイムスケジューリングアルゴリズムの研究も盛んに行われている．我々も過去の研究において，SMT プロセッサを対象としたリアルタイムスケジューリング方式である U-Link スケジューリングを提案している [9]．U-Link スケジューリングは，リアルタイム性及びスループットの点で従来のスケジューリング方式よりも効果的である．本論文では，U-Link スケジューリング方式を RMT Processor に実装し，理論的な優位性だけでなくスケジューラ呼び出しによるオーバーヘッドも削減できることも示す．また，マルチスレッドプロセッサでは，ユニプロセッサに比べて，スケジューリングアルゴリズムの性能がタスクセットに大きく依存する．そのため，目的に応じて複数のスケジューリングアルゴリズムを実装する必要性がでてきてる．そこで，複数のスケジューリングアルゴリズムにも対応できるように拡張性を考慮してリアルタイムスケジューラの設計及び実装を行う．

2. Responsive Multithreaded Processor

RMT Processor [8] は，優先度付き SMT やベクトル演算ユニットを中心とするリアルタイム処理用プロセッシングコアに加えて，リアルタイム通信規格 Responsive Link [10]，コンピュータ用 I/O(PCI64, USB2.0, IEEE1394 等)，制御用 I/O(PWM 発生器，パルスカウンタ等) を 1 チップに集積した並列分散リアルタイム処理用システム LSI である．

2.1 優先度付き SMT

RMT Processor では，SMT 実行によってシングルスレッド実行時の性能向上と複数スレッドの並列実行によるシステム全体の性能向上を実現している．しかしながら，SMT 実行では，スレッド間で演算器やキャッシュ等のハードウェア資源の競合が発生するので，各スレッドの実行効率が変動してしまう．リアルタイムシステムでは，予測性が非常に重要であるため，不規則な実行時間の変動が発生することは望ましくない．RMT Processor は，このような問題を軽減するために優先度付き SMT 実行を実現している．優先度付き SMT では，オペレーティングシステムが各スレッドに優先度を設定することを可能にし，優先度の高いスレッドに対して優先的にハードウェア資源を割り当てる．

RMT Processor は，8 つのハードウェアコンテキストを保持しているので，スレッド数が 8 以下の場合には，優先度付き SMT 実行によって静的優先度スケジューリングのようなリアルタイム実行が可能である．その様子を図 1 に示す．優先度の高いスレッド 1~3 の実行は，その他の低優先度スレッドの実行に影響を受けていないことがわかる．これにより，優先度の高いスレッドの実行効率の変動及び低下を抑制することが可能である．一方で，スレッドの優先度が低くなるにつれ，そのスレッド実行の実行に対する高優先度スレッドの影響が大きくなってしまうという問題点もある．そのため，RMT Processor の優先度付き SMT 実行だけでは，リアルタイム処理を実現することは困難である．しかしながら，優先度付き SMT 機構を効果的に扱えるスケジューリング機構があれば，RMT Processor を利用したリアルタイムシステムを実現することができると考えられる．

2.2 スレッド制御機構

RMT Processor においても，実行スレッド数が 8 を超えた場合にはコンテキストスイッチが発生する．一般的に，コンテキストスイッチは，スレッドの情報をメモリに格納する必要があるので，大きなオーバヘッドとなる．リアルタイムシステムにおいては，周期タスクを優先度に従ってスケジュールするため，コンテキストスイッチが頻繁に発生する．その際のオーバヘッドがリアルタイム性の低下につながることがしばしば問題とされる．RMT Processor では，コンテキストスイッチによる実行時オーバヘッドを削減するために，コンテキストを退避する専用のオンチップキャッシュであるコンテキストキャッシュを設けている．RMT Processor では 32 スレッド分のコンテキストをキャッシュでき，組み込み用途には十分であるといえる．

3. 設計及び実装

本節では，U-Link スケジューリング方式 (図 2) に Rate Monotonic(RM) [5] 及び Ealiest Deadline First(EDF) [5] を適用したアルゴリズムである UL-RM 及び UL-EDF の設計と実装に関して述べる．本論文では，簡単のため複合タスク [9] を用いないアルゴリズムを実装する (複合タスクを用いるアルゴリズムをそれぞれ UL-DRM 及び UL-DEDF と呼ぶ)．

3.1 スケジューリングアルゴリズム非依存部分

まず，スケジューリングアルゴリズムに依存した部分について説明する．ここで，紙面の都合上，排他制御やプリエンプション無効化等のコードは省略し，また，可読性を優先して表記していることに注意されたい．

実装対象である RMT Processor は，8 つのハードウェアコンテキストを保持しているが，命令発行幅が 4 であるため，本スケジューラでは論理プロセッサ (LP) は 4 つ用意した．すなわち，最高で 4 つのタスクが同時に実行可能である．各 LP は 1 つのレディキュー (RQ) を保持している．

3.1.1 タスク管理

RQ は，リスト構造として実装されることが多いが，リスト

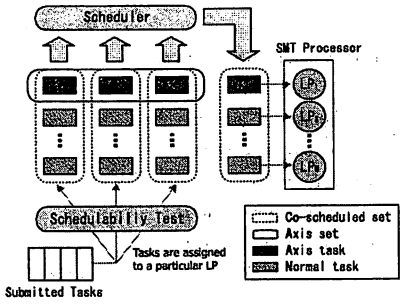

図 2 U-Link スケジューリング方式

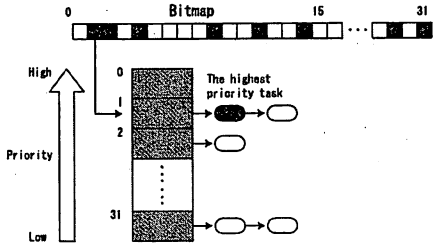

図 3 優先度配列

構造はタスク数が増えた時の計算量が大きくなってしまうという問題がある．一方で，リスト構造ではなく固定配列にしてしまうと，タスク数と同じ要素数が必要になってしまい，メモリの浪費になる．そこで，RQ の探索時間を削減し，かつメモリの浪費を避けるために，Linux の O(1) スケジューラで用いられている優先度配列 (図 3) を応用する．本研究では優先度レベルを 32 に設定した．

各タスクは優先度に応じて優先度配列の要素に割り当てられる．タスク数が増加すると，タスクの優先度が重なる場合が考えられるが，この際には，通常のリスト構造を使用して優先度順に連結すればよい．Bitmap は，優先度配列の要素にタスクが格納されている場合に対応するビットがセットされる．実際に，最高優先度タスクを取得する場合には，まず，一番最初にセットされているビットを得る．そして，そのビットに対応する優先度配列の要素であるリスト内で先頭のタスクが最高優先度タスクである．計算量は常に O(1) であり，ビット操作はシフト命令によって高速に行うことができるので，優先度配列によって RQ 操作のオーバーヘッドを削減できると考えられる．

次に，RQ へのタスクの追加及び削除方法について説明する．RQ の管理には，タスクを RQ に追加する enqueue_task 関数及び削除する dequeue_task 関数がある．

enqueue_task 関数は，スケジューリングアルゴリズム依存の task_prio 関数を用いて，優先度配列における優先度を算出する．次に，insert_array 関数を利用して優先度配列にタスクを格納する．最後に，set_bit 関数を用いて bitmap の対応するビットをセットする．一方，dequeue_task 関数は，list_del 関数を用いて，タスクを優先度配列の要素内リストから削除する．そして，そのリストが空になった場合には，bitmap の対応するビットをクリアし，その要素にはタスクが格納されていないことを明示する．スケジューリングアルゴリズムを実装する際には，task_prio 関数を用意するだけで済み，拡張性を考慮した設計であることがわかる．

```
void enqueue_task(*task, *rq)
{
  task->priority = task_prio(task, rq);
  insert_array(task, rq->array + task->priority);
  set_bit(task->priority, rq->bitmap);
  task->rq = rq;
}
```

図 4 enqueue_task 関数

```
void dequeue_task(*task, *rq)
{
  list_del(&task->run_list);
  if (list_empty(rq->array + task->priority))
    clear_bit(task->priority, rq->bitmap);
}
```

図 5 dequeue_task 関数

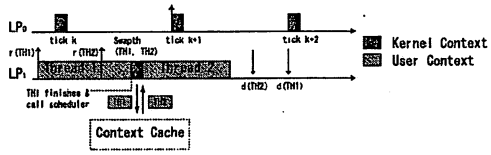

図 6 明示的呼び出し

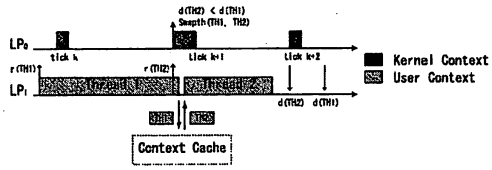

図 7 暗黙的呼び出し

3.1.2 タスクスケジューリング

スケジュール関数が呼ばれるタイミングは 2 通りある．1 つは，あるタスクの実行が終了した時にそのタスクによって呼び出される場合である．もう 1 つは，一定の間隔 (system tick) で起動するタイマー処理によって，新しいタスクがリリースしていた場合のみ呼び出される場合である．前者を明示的呼び出し (図 6)，後者を暗黙的呼び出し 7) と定義する．

```
void task_period_end(*task)
{
  rq = task->rq; lp = task->run_lp;
  dequeue_task(task, task->rq);
  task_period_end_algo(task);
}
```

図 8 task_period_end 関数

```
void global_schedule(*rq)
{
  temp[0...3] = 4 of highest priority tasks;
  for each lp = LP[0...3] {
    prev = lp->current_task;
    next = task[j++];
    next->run_lp = lp;
    context_switch(prev, next);
    lp->current_task = next;
  }
}
```

図 9 global_schedule 関数

task_period_end 関数は，毎回の周期でタスクの実行が終了した場合に呼び出される関数である．タスクを RQ から削除し，スケジューリングアルゴリズム依存の task_period_end_algo 関数を呼び出す．明示的呼び出しを行う場合は，task_period_end_algo 関数内で行う必要があり，スケジューリングアルゴリズムに依存することになる．

本論文では，拡張性の高いスケジューラを目標としているため，U-Link スケジューリング方式だけなくグローバルスケジューリング方式及びパーティショニング方式にも対応する．そのため，スケジュール関数は global_schedule 関数 (図 9) と local_schedule 関数 (図 10) の 2 種類を提供する．global_schedule 関数は，4 つの優先度の高いタスクを大域的に選択し，各 LP で実行する．4 つのタスクの中で，既に実行しているタスクに関しては実際のコンテキストスイッチは発生しない．一方，local_schedule 関数は比較的単純であり，指定された RQ から最も優先度の高いタスクを選択し，指定された LP で実行する．

グローバルスケジューリング方式では，暗黙的呼び出しで global_schedule 関数を実行し，明示的呼び出しによって lobal_schedule 関数を実行する．一方，パーティショニング方式は，暗黙的呼び出し及び明示的呼び出しの場合でも local_schedule 関数を実行する．U-Link スケジューリングでは，特定の LP で軸タスクのみスケジュールするので，パーティショニング方式と同様に local_schedule 関数のみを利用する．

```
void local_schedule(*lp, *rq)
{
  next = highest priority task;
  prev = lp->current_task;
  next->run_lp = lp;
  context_switch(prev, next);
  lp->current_task = next;
}
```

図 10 local_schedule 関数

```
void task_prio(*task, *rq)
{
  if (task->period > rq->max)
    return PRIO_NUM;
  else if (task->period < rq->min)
    return 0;
  else
    return (PRIO_NUM * (task->period - rq->min))
      / (rq->max - rq->min);
}
```

図 11 task_prio 関数 (UL-RM)

```
void task_prio(*task, *rq)
{
  return 0;
}
```

図 12 task_prio 関数 (UL-EDF)

3.2 アルゴリズム依存部分

アルゴリズム依存部分で最も重要なのがタスクの優先度を決定する task_prio 関数である．UL-RM では，優先度配列を有効利用した task_prio 関数を用意した (図 11)．UL-RM は，軸タスクに関してのみスケジュールを行い，そのスケジュールは従来の RM と全く同じである．まず，タスクの周期が優先度配列内の最大タスク周期よりも大きければ，最も低い優先度を与える．一方，優先度配列内の最小タスク周期よりも小さければ，最高優先度を与える．それ以外の場合では，最大周期と最小周期の差から，タスクの周期に対応した優先度を設定する．UL-EDF は，優先度が動的に変動するため，優先度配列を用いると優先度の再計算のオーバーヘッドが大きくなってしまう．そこで，全てのタスクに同じ優先度を与え，簡単なリスト構造によってタスクを管理する (図 12)．

```
void context_switch(*prev, *next)
{
  for (;;) {
    if (!prev && next)
      context_run(next);
    else if (prev && !next)
      context_stop(prev);
    else if (prev && next)
      context_swap(prev, next);
    else
      break;
    prev = prev->cosched_next;
    next = next->cosched_next;
  }
}
```

図 13 context_switch 関数

U-Link スケジューリングでは，同時実行セット内のタスクを常に軸タスクに同期して実行する必要がある [9]．その作業は，context_switch 関数 (図 13) で行う．

context_switch 関数は，循環リスト prev->cosched_next に繋がっている全てのタスクの実行を停止し，循環リスト next->cosched_next に繋がっている全てのタスクの実行を開始する．これにより，同時実行セット内のタスクを常に同時に実行することができる．しかしながら，実行の開始だけでなく実行の終了も同期しなくてはならない．さもなければ，前のタスクが実行しているにもかかわらず，次のタスクを実行するような事態が発生する．

実行終了の同期は，task_period_end_algo 関数内で行う (図 14)．タスクの同期にはマシン依存関数である barrier 関数を利用する．実行している全てのタスクが barrier 関数を呼び出したら，実行が再開される．ここで，軸タスクのみが schedule 関数を呼び出し，先に示した context_switch 関数によって同時に実行を再開する．highest 関数及び middle 関数は，タスクの重要度を設定する関数である．RMT Processor においては，ハードウェアスレッドの優先度を設定することで，スケジューラの実行に常に優先してハードウェア資源を割り当てることを可能にしている．これにより，スケジューラの処理を速やかに終えることができる．

グローバルスケジューリング方式の欠点は，暗黙的呼び出しにおいて 1 つのスケジューラが全てのタスクを管理しなければならないため，オーバーヘッドが大きいことである．一方，パーティショニング方式は，各 LP でタスクが分散され，独立にスケジューリングするため，暗黙的呼び出しにおけるオーバーヘッドは小さい．しかしながら，グローバルスケジューリング方式は，スケジューラ専用の論理プロセッサを別途に用意して，タスクの実行と並列にスケジューラを起動することができる．これにより，オーバーヘッドを隠蔽することができるが，パーティショニング方式ではスケジューラが各 LP に 1 つ必要になるので，別途用意することはハードウェア資源の点で難しい．U-Link スケジューリング方式は，スケジューリングの対象は軸タスクのみなので，パーティショニング方式と同様にオーバーヘッドは少ない．さらに，スケジューラは 1 つしか存在しないので，グローバルスケジューリング方式のようにスケジューラ専用の LP を用意することが可能である．すなわち，U-Link スケジューリング方式は，パーティショニング方式とグローバルスケジューリング方式の長所を兼ね備えている．また，表 1 にあるタスクが与えられた時，各スケジューリング方式は図 15~17 に示すようにスケジュールを行う．U-Link スケジューリング方式は，スケジューラの明示的呼び出しの回数がグローバルスケジューリング方式及びパーティショニング方式に比べて少ないことがわかり，スケジュール呼び出しによるオーバーヘッドを削減できる．図の例は LP 数が 2 の場合であるが，LP 数が多くなれば，その効果も大きくなることが予想される．

```
void task_period_end_algo(*task)
{
  barrier(nr_running);
  if (task is the axis) {
    highest(task);
    lobal_schedule(task->run_lp, task->rq);
    middle(task);
  }
}
```

図 14 task_period_end_algo 関数

表 1 タスクセット例

Task	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
最悪実行時間	1	3	3	2	2	2
周期	5	10	10	10	5	5

4. 評　　価

RMT Processor の性能は文献 [9] と同様に設定した．

まず，優先度配列を応用して RQ を実装した場合と単純なリスト構造を用いて RQ を実装した場合に，タスクの RQ への追加に要す時間を計測し，比較を行った (図 18)．タスク数が 15 以下であればリスト構造の方が短い時間でタスクを RQ へ格納できている．しかしながら，それ以上タスク数が増加すると，リスト構造での RQ 操作時間は増し，最終的には 20.5μs 要した．一方で，優先度配列を用いた場合には，タスクの RQ への追加に要す時間は，ほぼ一定であり，7.0μs ~ 8.5μs であった．この結果から，リスト構造はタスク数が多ければ多いほど，RQ の操作に時間を要し，優先度配列はタスク数に関係なく一定の

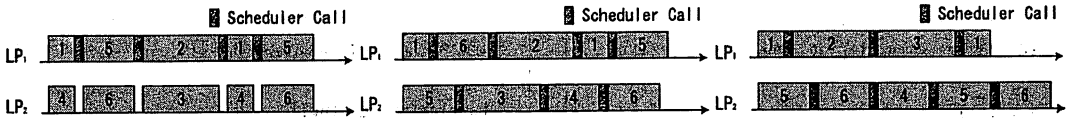

図 15 U-Link スケジューリング方式でのスケジューラの明示的呼び出し

図 16 グローバルスケジューリング方式でのスケジューラの明示的呼び出し

図 17 パーティショニング方式でのスケジューラの明示的呼び出し

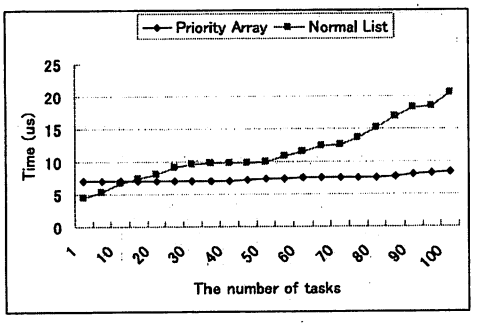

図 18 タスクの RQ への追加に要す時間

表 2 スケジューラの呼び出しに関する評価 (RM)

	UL-RM	RM-US	RM-FF
呼び出し回数比	1	3.5	3.9
1 回のオーバーヘッド	12.45μs	29.34μs	12.32μs

表 3 スケジューラの呼び出しに関する評価 (EDF)

	UL-EDF	EDF-US	EDF-FF
呼び出し回数比	1	3.4	3.5
1 回のオーバーヘッド	25.30μs	41.00μs	23.29μs

時間を要すことがわかった．最終的に，優先度配列はリスト構造よりも 40% ほどオーバーヘッドを削減したことになる．

次に，U-Link スケジューリング方式，グローバルスケジューリング方式，パーティショニング方式のスケジューラ呼び出し回数及び 1 回のスケジューラ呼び出しに要する最悪のオーバーヘッドに関する評価を行った．タスクセットは文献 [9] と同様に生成するがタスクの周期はハーモニックになるようにし，デッドラインミスが起こらないようにシステムの負荷が 150% になるように設定した．また，システムの実行時間は 8ms とした．RM をベースとしたアルゴリズムは，それぞれ，UL-RM, RM-US [1], RM-FF [2] である．一方，EDF をベースとしたアルゴリズムは，それぞれ，UL-EDF, EDF-US [4], EDF-FF [6] である．評価結果を表 2 及び表 3 に示す．ここで，各アルゴリズムの評価値は UL-RM 及び UL-EDF の回数で正規化したものである．この結果から，U-Link スケジューリング方式のスケジューラ呼び出し回数がベースとなるアルゴリズムにかかわらず最も低いことがわかった．先にも述べたように，U-Link スケジューリング方式は，軸タスクがコンテキストスイッチを行う時のみスケジューラを呼び出すため，その呼び出し回数が大幅に削減されたと考えられる．また，スケジューラが 1 回呼び出された時のオーバーヘッドは，RM 及び EDF 共にパーティショニング方式が最も小さかった．U-Link スケジューリング方式はオーバーヘッドはわずかに大きかったが，ほぼ同等であるとみなせる．グローバルスケジューリング方式は，2 倍以上のオーバーヘッドが計測された．これは，global_schedule 関数が 4 つのタスクを処理する必要があるのに対して，local_schedule 関数では 1 つのタスクしか処理しないためであると考えられる．

5. 結　論

本論文では，RMT Processor を対象としたリアルタイムスケジューラの設計及び実装について述べた．拡張性という点では，スケジューリングアルゴリズムに依存しない部分と依存する部分に切り分け，部分間のインターフェイスを定義した．オーバーヘッドの点では，タスク管理機構に優先度配列を応用し，タスクの RQ への追加に要す時間を削減した．また，U-Link スケジューリング方式の実装では，グローバルスケジューリング方式及びパーティショニング方式に比べて，スケジューラの呼び出しによるオーバーヘッドを削減することができた．

謝辞 本研究は，科学技術振興機構 CREST の支援による．

文　献

[1] B. Andersson, S. Baruah, and J. Jonsson. Fixed-priority scheduling on multiprocessors. In *Proc. of International Conference on Real-Time Systems Symposium*, pages 193–202, 2001.

[2] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26:127–140, 1978.

[3] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17:12–19, 1997.

[4] J. Goossens, S. Funk, and S Baruah. Priority-driven scheduling of periodic task systems on multiprocessor. *Real-Time Systems*, 25:187–205, 2003.

[5] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of ACM*, 20:46–61, 1973.

[6] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28:39–68, 2004.

[7] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[8] N. Yamasaki. Responsive multithreaded processor for distributed real-time systems. *Journal of Robotics and Mechatronics*, 17:130–141, 2005.

[9] 加藤真平, 小林秀典, and 山崎信行. Smt プロセッサを対象としたリアルタイムスケジューリング. In **コンピューティングシステムシンポジウム**, pages 109–118, 2005.

[10] 山崎 信行. 分散制御用リアルタイム通信 Responsive Link の設計および実装. **情報処理学会論文誌コンピューティングシステム**, 45(SIG 3):50–63, 2004.